A framework for automatic and parameterizable memoization

Loïc Besnard, Pedro Pinto, Imane Lasri, João Bispo, Erven Rohou, João Manuel Paiva Cardoso

A Framework for Automatic and Parameterizable Memoization

Loïc Besnard[a], Pedro Pinto[b], Imane Lasri[a], João Bispo[b], Erven Rohou[a], João M. P. Cardoso[b]

[a] *Univ Rennes, Inria, CNRS, IRISA Campus de Beaulieu, 35042 Rennes France. Email: loic.besnard@irisa.fr, {imane.lasri, erven.rohou}@inria.fr*

[b] *Faculty of Engineering, University of Porto, Porto, Portugal. Email: {p.pinto, jbispo, jmpc}@fe.up.pt*

Abstract

Improving execution time and energy efficiency is needed for many applications and usually requires sophisticated code transformations and compiler optimizations. One of the optimization techniques is memoization, which saves the results of computations so that future computations with the same inputs can be avoided. In this article we present a framework that automatically applies memoization techniques to C/C++ applications. The framework is based on automatic code transformations using a source-to-source compiler and on a memoization library. With the framework users can select functions to memoize as long as they obey to certain restrictions imposed by our current memoization library. We show the use of the framework and associated memoization technique and the impact on reducing the execution time and energy consumption of four representative benchmarks.

Keywords: memoization, compiler optimizations, source-to-source

1. Motivation and significance

Improving execution time and energy efficiency is of paramount importance in many applications. A possible optimization that can be used for such improvements is memoization [1], which is the technique of saving the results of computations so that future computations can be skipped. Performance can be improved by caching execution results of memoizable functions, and retrieving them instead of recomputing a result when a new call is performed with repeated inputs. In this work we introduce a memoization framework, integrated in the ANTAREX tool flow [2], which can automatically apply this technique to memoizable functions.

We define memoizable functions [3, 4] as pure functions, i.e., deterministic and without side effects, with three additional constraints imposed by our current implementation of memoization. First, functions must have between 1 and 3 parameters of the same type T. This limitation arises from our current choice of hash function, which combines all the arguments and, as such, requires them to have the same bit length. After combining all arguments with *XOR* operations, we take the high-order and the low-order bits of the result and combine them again using a *XOR*, leaving us with half the original bits. We repeat this process until we arrive to the number of bits required to index the hash table, masking any bits if necessary. The second constraint is that functions must return data of the same type T. Finally, the third constraint is that T is one of `int`, `float` or `double`.

In applications with memoizable functions in critical code sections, memoization may provide important execution time reductions and energy consumption savings. Our memoization framework relies on internal tables that store the results of previous computations and replace future calls by table lookups. The elements of the internal tables are indexed with a hash calculated from the call arguments of the memoized function.

A memoization approach can start by profiling the application and by identifying the contribution of memoizable functions to the overall execution of the application. A profiling step may also provide values to setup an initial version of the internal table of the memoization technique. Our framework allows loading internal tables before application runs (e.g., with profiling data) and/or update them at runtime (allowing adaptation to execution contexts not considered during the profiling phase). This solution may enable savings in multiple kinds of applications and allow users to write runtime adaptivity strategies to make applications more resilient to context changes and able to achieve predetermined execution thresholds.

The memoization technique can be integrated into any C or C++ application that has memoizable functions or methods. Based on the selected memoizable functions, our framework generates a new version of the application enhanced with memoization support by relying on the Clava[1] source-to-source compiler. The framework is also responsible to generate the C library that contains the core of the memoization implementation, which is linked with the newly generated application. The compilation process of Clava is controlled by the LARA [5] Domain-Specific Language (DSL). With this setup, it is also possible to use LARA to perform analyses and transformations on the code, e.g., where and how to apply the memoization technique in

[1] `https://github.com/specs-feup/clava`

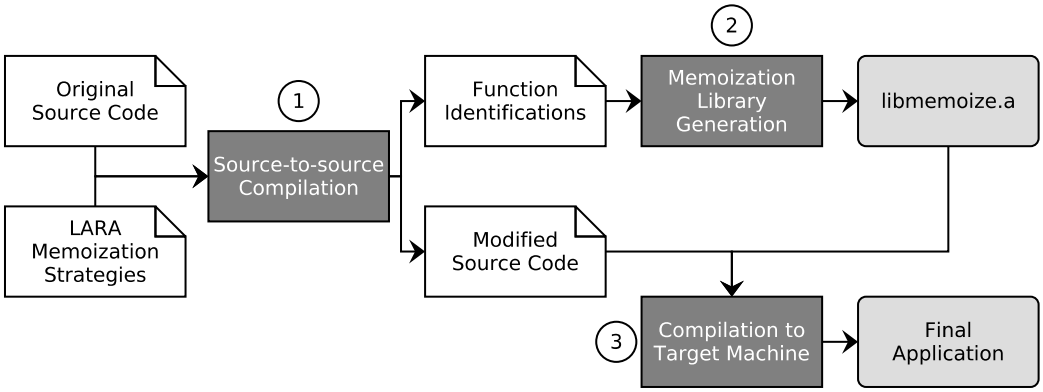

Figure 1: Tool flow of the proposed memoization framework.

this specific case. The main modular unit in LARA code is the *aspect*, which can be considered as a function, with input and outputs, but that has the capability of interacting with the internal representation of the application source code. When several aspects are developed to achieve a specific goal, we have a LARA program, which we often refer to as a *strategy*.

Our memoization framework is based on the memoization approach presented in [4], that uses a dynamic library and applies memoization at binary level. Our framework has been generalized for C++, adds extensions that allow application flexibility, adds options regarding table loading and table runtime updating, and is applied at the level of the application source code.

2. Software description

The software solution presented in this paper relies on two different components which are combined to easily enhance an application with memoization support. The first component is Clava, a source-to-source compiler able to automatically generate a new version of the application code with the necessary changes to support memoization. This is performed through a series of LARA aspects the user can call and parameterize, all distributed as part of the Clava memoization library.

The second component is the memoization C library itself, which is generated from a configuration file that identifies the functions on which to apply memoization. The library is compiled and linked against the version of the input application that has been modified by Clava. As shown in Figure 1, the use of this memoization solution consists of three steps: 1) source-to-source compilation; 2) generation and compilation of the memoization library; and 3) compilation of the application linked with library.

```c
float foo (float p)
{
  /* code of foo without side effects */
}

float foo_wrapper(float p)
{
  float r;

  /* already in the table ? */
  if (lookup_table(p, &r)) return r;

  /* calling the original function */
  r = foo(p);

  /* updating the table or not */
  update_table(p, r);

  return r;
}
```

Figure 2: A memoizable C function and its wrapper.

Consider the memoizable C function `foo` shown in Figure 2. This is a pure function which takes a single `float` parameter and returns data of the same type. The source-to-source step consists of two phases, with the first phase being the addition of another function to the program. This new function *wraps* the target function and includes the memoization logic to interface to the memoization library. This wrapper is also illustrated in Figure 2. The second phase replaces all calls (it is possible to specify a LARA aspect that applies memoization to only certain call sites) to the original function, `foo`, with calls to its corresponding wrapper, `foo_wrapper`. This technique works for C and C++ functions as well as for C++ methods, and takes into account name mangling, function overloading, and references to objects.

The user can, through LARA aspects or manually, further change the application to interact with the memoization library. For instance, the memoization library exposes a set of variables to control its internal behavior. To dynamically stop and restart the memoization for a specific function, it is possible to change the variable `_Memoize__<mname>`, where `<mname>` corresponds to the mangled name of the target function. Similarly, to control the table update policies the user can change the variables `_alwaysReplace__<mname>` and `_FullyOffLine__<mname>`. This can be used to control the runtime behavior of the memoization library with respect to the update of the internal table.

The generation and compilation of the memoization library start from the function identifications file (`funs-static.def`). The LARA library for

memoization, besides changing the source of the application, generates this file. In this article, we propose the use of memoization through Clava, by relying on aspects programmed with the LARA DSL. However, it is also possible to use the standalone library[2] without relying on LARA and the Clava compiler. To do so, the user must manually write the function identifications file (`funs-static.def`), change the source code to include the wrappers and replace calls from the original calls to the wrappers. The rest of the flow remains the same: generate the library and link it when compiling the application.

The advantage of using LARA strategies is that the memoization library is integrated into the application without performing manual modifications of the source code. The code generated by Clava is then compiled and linked with the associated generated memoization library.

In order to enhance an application with memoization with LARA strategies, users can call the library aspects and define a number of parameters. First, it is possible to define the size of the internal table that stores the computation data. This decision becomes a tradeoff that can be fine-tuned according to the needs of the current application or scenario. A larger table holds more data but requires more memory. On top of that, depending on the underlying architecture, the size of the table may impact cache performance. Second, one can specify a file from which to load a previously initialized memoization table. If none is specified, the program starts with an empty table. This allows building memoization tables from one execution to the following and keep the most frequently requested outputs. For this feature, we can also specify a file on which the results of the execution are saved. Finally, if one intends to use the approximation capabilities, it is also possible to configure the library at this step to disregard a number of bits from the input representation of the arguments to the memoized function.

2.1. Technique Details

The first operation of the wrapper is to perform a lookup to check whether the output for the provided input is already stored. The lookup operation will hash the input to get the correct slot in the internal table. If this position is empty, the lookup fails and returns `false`, signaling a miss. If this happens, the original target function is called and the correct value is returned. On the other hand, if the table slot is not empty, we compare the stored input with the input of the current call. If these values are the same, which we consider a hit, the lookup function sets the output value and returns successfully. The

[2] `https://gforge.inria.fr/projects/memoization`

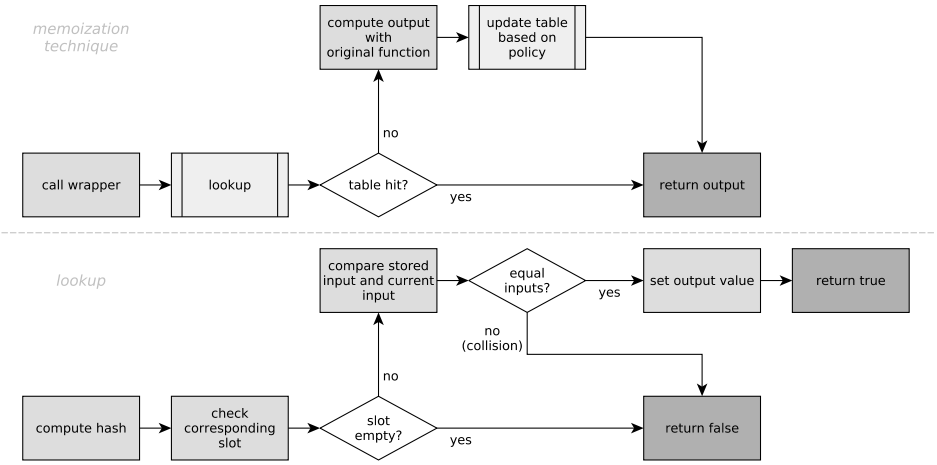

Figure 3: Flowcharts describing the overall memoization technique and the lookup process.

wrapper will then return this value. If the inputs are not equal we have a collision, meaning that two different inputs were hashed into the same slot. In this case, the lookup function returns `false`. Once again, the return value is computed by calling the original function, which is eventually returned. The correct value is always returned whether there was a hit, a miss or a collision. The table update logic happens after the correct output value has been computed (or fetched from the table), so it is inconsequential to the safeness of the technique. Figure 3 shows two flowcharts describing the overall memoization technique (on top) and the lookup process (on bottom).

2.2. Software Architecture

Figure 4 shows an overview of the architecture our memoization framework. The four main components can be divided into two groups, *generation* and *execution*. The generation group, with Clava and the Library Generator, is responsible for generating the execution group, composed by the augmented application code and the memoization library. The dashed arrows indicate that one component generates another and the solid arrows indicate runtime interactions between components.

Clava is a source-to-source compiler, written in Java, that uses Clang[3] as a front-end to parse C and C++ source code. The compiler maintains an internal representation of the application which is analyzed and transformed according to the strategies defined in the input LARA code. Clava includes

[3] `https://clang.llvm.org/`

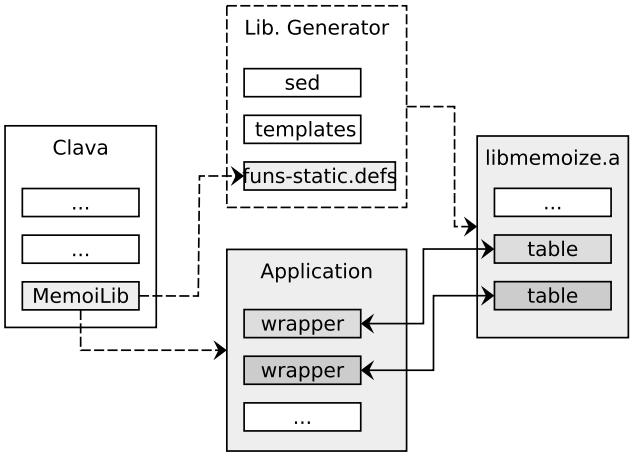

Figure 4: Main components of the proposed memoization solution.

multiple LARA libraries that can be used for different purposes, including the memoization framework presented in this article.

The Library Generator is a `sed` script that takes the function definitions file (`*.def`) and several code templates to generate the final customized code of the memoization library. This code can then be compiled into a library. The library itself maintains a table for each memoized function as well as the associated variables for runtime control and the interface functions. These are the functions called from each of the wrappers inserted with the Clava memoization strategies.

The final application retains the original functionality. In the source code, the difference is that it has a wrapper for each target function. This wrapper is called instead of the original and performs calls to the generated memoization library.

2.3. Software Functionalities

The memoization framework presented in this paper allows users to semi-automatically enhance an application with memoization support. The typical user intervention is related to the configuration options and to the selection of the functions to memoize. However, function selection can be also performed without user intervention as the LARA library also provides code to automatically detect memoizable functions and methods. This option returns the list of detected memoizable functions and users can decide whether to apply memoization on these returned functions.

```
import antarex.memoi.Memoization;

aspectdef Launcher
        call Memoize_Initialize( );
        call Memoize_MathFunctions(['cos' , 'acos' , 'sqrt']);
        call Memoize_Function('myfunc');
        call Memoize_Finalize( );
end
```

Figure 5: An example of LARA Launcher aspect defined for memoization.

The LARA library integrated into the Clava compiler has several aspects that are used to apply memoization to different types of functions: mathematical functions (from `math.h`), C functions and C++ functions. For each of these, there are LARA aspects that allow the user to memoize a function with default parameters or by defining all the needed parameters. Additionally, the library includes an aspect that automatically performs memoization on all `math.h` functions, and aspects that apply memoization to C++ functions and methods taking into account function overloading.

Since our framework includes source-to-source compilation, it is possible to write sophisticated analysis and transformation strategies using LARA aspects. This can be used, for instance, to select which functions to memoize using any user-devised heuristics on top of the list of possible memoizable function returned by the library.

Another feature of our memoization library is the possibility of using approximation when dealing with floating point numbers. This essentially removes a number of (user-specified) bits from the inputs to the function, grouping together close numbers and returning the same output for those. More precisely, we remove a number of the least significant bits of the mantissa. In scenarios where some loss of accuracy is tolerable (e.g., in some cases of image processing) this can provide further reductions in execution time and energy consumption.

All of these features enhance an application with the capability of taking advantage of repeating inputs on some specific and critical functions, while still being able to accommodate changes, since the table can be dynamically updated. Furthermore, users can explore runtime strategies to decide when to change the update policies of the table or completely turn memoization off, thanks to the variables that expose this type of control for each target function.

```
1 DEF(0, acos, acos_wrapper, 1, double, 0, none, no, none, no, 65536)
2 DEF(0, cos, cos_wrapper, 1, double, 0,  none, no, none, no, 65536)
3 DEF(0, sqrt, sqrt_wrapper, 1, double, 0,  none, no, none, no, 65536)
4 DEF(2, myfunc, myfunc_wrapper, 3, double, 0, none, no, none, no, 65536)
```

Figure 6: An example of funs-static.def file.

3. Illustrative Examples

This section presents an example of a LARA aspect that illustrates the interface between the user and the memoization library described in the previous section.

Consider an example of a C application that uses the mathematical functions `cos`, `acos` and `sqrt`. Moreover, the profiling of the application shows that there is a large number of calls to a user function called `myfunc`. Figure 5 shows a LARA aspect, named *Launcher*, that can be used to apply memoization. First, we import the class that contains the memoization aspects (line 1). Then, the functions to be memoized are identified (lines 5-6), preceded by an initialization (line 4) and followed by a finalization stage (line 7). This is all the LARA code user needs to write to enhance a C/C++ application with memoization.

The execution of the aspect will produce (1) a new version of the application in which all the references to FOO (FOO is one of `cos, acos, sqrt, myFunc`) are replaced by FOO_wrapper, and (2) the `funs-static.def` file that contains a line per memoized function as shown in Figure 6. Each line encodes all the required parameters for the generation of the memoization C library.

The `Memoize_Function` aspect is a helper aspect that automatically sets some default parameters when calling other, more general aspects. For instance, line 4 of Figure 6 exposes some of the parameters that can be specified on those more general aspects. It specifies that it is a C user function (value 2), followed by the name of the function (`myfunc`) and its associated wrapper (`myfunc_wrapper`), and the identification that this function has 3 inputs arguments of `double` type. Other parameters are provided to control the memoization framework: in particular the 2 last ones (`no, 65536`) are used to specify the policy to manage possible conflicts when updating the table (replace or not) and to define the size of the internal table. LARA aspects are provided to the user to set these parameters, but in the presented example we call simpler versions that use default values.

For example, Figure 7 shows an example of the implementation of one of the LARA aspects that are part of the memoization library. This is library

```
1 aspectdef Memoize_Method_overloading_ARGS
2 input
3   aClass,      // Name of a class
4   aMethod,     // Name of a method of the class aClass
5   pType,       // Name of the selected type
6   nbArgs,      // Number of parameters of the method
7   fileToLoad, // filename for init of the table, or 'none'
8   FullOffLine,// yes/no. yes for a fully offline strategy
9   FileToSave, // filemane to save the table, or 'none'
10  Replace,     // yes/no. yes: always replace in case of collisions
11  approx,      // Number of bits to delete for approximation.
12  tsize        // Size of the internal table.
13 end
14 // Control on the parameters of the aspect: nbArgs in [1,3]
15    ...
16 // Searching the method.
17 var MethodToMemoize, found=false;
18 select class{aClass}.method{aMethod} end
19 apply
20  if (! found) {
21    found = isTheSelectedMethod($method, nbArgs, pType);
22    if (found) MethodToMemoize=$method;
23  }
24 end
25 if (!found)
26  { /* message to the user */}
27 else {
28  GenCode_CPP_Memoization(aClass, aMethod, pType, nbArgs,
29  fileToLoad, FullOffLine, FileToSave, Replace, approx, tsize);
30  call CPP_UpdateCallMemoization(aClass, aMethod, pType, nbArgs);
31 }
32 end
```

Figure 7: An example of LARA aspect defined for C++ memoization.

code and not the code a user needs to write. This is used to target C++ methods and allows the user to specify all memoization parameters. This example defines the memoization (lines 1-13) of a C++ method (`aMethod`) of a class (`aClass`) with `nbArg` parameters of the same type as the returned type (`pType`).

After some verifications, which are not presented in this example, the target method is searched in lines 17–24. In case of success, the code of the wrapper is added (line 28) to produce the memoization library, and the code of the application is modified for calling the generated wrapper (line 30), which is also declared as a new method of the class.

4. Impact

In order to show the impact of our memoization framework we have applied it to four C/C++ programs. For each program we have tested the

original version as well as four different memoized versions representing a combination of two parameters. The first parameter is the internal table size and we tested with tables holding 256 and 65,536 elements. The second parameter is whether to perform table updates in case of hash collisions. For this experiments we do not consider the setup of tables prior to the execution of the application. Thus, when the memoization-enhanced program starts executing, its tables are empty and start being filled as requests are performed. It is possible that two different inputs for the same function produce the same hash value, which leads to a collision. Our strategy in this example is to either ignore the collision and not change the table, or replace the previous entry with the newest one.

Our benchmark applications are:

- **atmi** [6] is a library for modeling steady-state and time-varying temperature in microprocessors, with examples using `j0`, `j1`, `exp` and `sqrt`.
- **equake** is an application extracted from SPEC OMP. It has calls to the mathematical functions `sin`, `cos` and `sqrt` in its critical region.
- **fft** is a Fast Fourier transform implementation extracted from the BenchFFT[4] benchmark suite. It calls the functions `sin` and `cos`.
- **rgb2hsi** is a benchmarking kernel that converts images from RGB model to HSI model. It calls `cos`, `acos`, `sqrt`, and a pure user function.

The tests have been performed on an Intel(R) Core(TM) i7-5600U CPU @ 2.60GHz and the C/C++ codes were compiled using GCC with -O3.

To address possible concerns that this technique may be too dependent on which inputs it is tested on, we have used several different inputs for the **fft** and **rgb2hsi** benchmarks. For **fft**, we tried three numbers of samples (100000000, 200000000, 300000000) and three frequencies (123.6, 256.89, 88.7). For **rgb2hsi**, we used eight different images. The other benchmarks were tested with a single input due to the lack of available workloads.

Table 1 presents the speedups obtained with the four tested memoziation configurations, over the original version (i.e., without memoization). Table 2 presents, for the same configurations, the energy consumption improvements over the original. This was measured using a utility tool that relies on Intel RAPL counters. Values below 1 represent a reduction in energy consumption. For instance, for the **atmi** benchmark with a *65536-nu* configuration, we only consume 71% of the original (0.71 in the table).

[4] `http://www.fftw.org/benchfft`

Overall, the use of our memoization framework allows us to achieve considerable reductions in both execution time and energy consumption. This is confirmed with the *Mean* line, which presents the geometric mean across benchmarks of the results obtained with each tested configuration. Considering execution time, the worst average result is a 0.97× slowdown when using a small table and no update (in the case of collisions). On the other hand, a larger table with updates enabled, achieves an average speedup of 1.16×.

With respect to the **atmi** and **rgb2hsi** benchmarks, a larger table has a positive impact, while this does not happen in the other two benchmarks. **atmi** and **rgb2hsi** show only a small number of hits (i.e., results found on the table) on the 256 table, which grows considerably when the table size is increased, while on the **equake** and **fft** benchmarks this growth is not as large. This can happen because of the way inputs to the memoized functions change over time and how many different values these inputs take.

We note that enabling update in the case of hash collision improves the obtained results. This is the case with every benchmark tested, except with **atmi** in the case of using a large table. When updating is disabled, it is possible to initially fill the table with inputs that are among the least frequently requested during the rest of the execution. This is somewhat corrected by turning on the ability to update in case of collision, since most frequent inputs should be able to push out other, less-frequent, inputs and remain in the table for most of the execution. However, in the specific case of **atmi**, the Bessel functions show an increasing number of collisions when updating is enabled, which seems to indicate that two frequent inputs are colliding and continually pushing each other off the table, causing misses and forcing the recalculation of their corresponding outputs.

The **fft** benchmark shows a performance degradation when using memoization without updates, regardless of the table size. When looking at more detailed execution data (not presented here), it is possible to see that there is a single hit for `cos` (out of 200,000,000 calls) and 0 hits for `sin` (out of 100,000,000 calls). This means the tables are initially filled with inputs that are not requested again in the future, causing an overhead which slightly degrades the original performance. On the other hand, when the updates are enabled, these initial values are eventually overwritten by more frequent inputs. The number of hits increases to 137,227,265 and 74,893,090 for `cos` and `sin`, respectively.

The **rgb2hsi** benchmark shows a severe drop in performance when using a small table without update on collisions. This benchmark benefits greatly from both a larger table and updates to remove less frequent values from the table. The number of hits increases by over 6× by simply using the larger table, in the case of no update. This may happen when inputs are distributed

Table 1: Speedups of the memoization versions over the original version. Each of the four columns corresponds to memoized benchmarks with two different parameters, table size and collision update policy (*nu* for no update, and *u* for update). The last line of the table presents the geometric mean of the speedups per configuration.

Benchmark	256-nu	256-u	65536-nu	65536-u
atmi	1.01	1.02	1.46	1.26
equake	1.02	1.06	1.01	1.04
fft	0.98 ± 0.01	1.13 ± 0.02	0.98 ± 0.02	1.12 ± 0.02
rgb2hsi	0.89 ± 0.02	1.13 ± 0.10	1.18 ± 0.06	1.22 ± 0.08
mean	0.97 ± 0.06	1.08 ± 0.05	1.14 ± 0.22	1.16 ± 0.10

Table 2: Energy consumption improvements of the memoization versions over the original version. Each of the four columns corresponds to memoized benchmarks with two different parameters, table size and collision update policy (*nu* for no update, and *u* for update). Values below 1 represent a reduction in energy consumption. The last line of the table presents the geometric mean of the improvements per configuration.

Benchmark	256-nu	256-u	65536-nu	65536-u
atmi	1.02	1.01	0.71	0.83
equake	0.96	0.93	0.95	0.94
fft	1.03 ± 0.03	0.89 ± 0.03	1.04 ± 0.03	0.90 ± 0.03
rgb2hsi	1.20 ± 0.03	0.95 ± 0.08	0.88 ± 0.04	0.86 ± 0.05
mean	1.05 ± 0.11	0.94 ± 0.05	0.89 ± 0.14	0.88 ± 0.04

in such a way that most calls use the same set of inputs, but this set is large enough so that it cannot fit into a small table.

In practice we do not frequently use tables as small as 256 elements. However, this illustrates the tradeoffs a user can perform to fine tune this framework to the current needs. In some scenarios, such as when targeting embedded systems with limited resources, a smaller table could be needed due to memory restrictions.

Table 3 presents the results of a two-tailed, paired-sample *t-test* performed on the execution time measurements of the **fft** and **rgb2hsi** benchmarks (the two benchmarks tested with multiple inputs). Tests on the energy results follow the same pattern. This statistical test was applied to compare the results of each memoization configuration against the results of the original version (i.e., without memoization), when running with different inputs (9 **fft** configurations and 8 **rgb2hsi** images). There is a single configuration (highlighted in the table) on which we cannot claim statistical significance

Table 3: Results of a two-tailed, paired-sample *t-test*.

Benchmark	256-nu	256-u	65536-nu	65536-u
fft ($df = 8$)	$t = 3.50$ $p = 0.008$	$t = 6.45$ $p = 0.000$	$t = 2.91$ $p = 0.020$	$t = 6.01$ $p = 0.000$
rgb2hsi ($df = 7$)	$t = 3.63$ $p = 0.008$	$t = 2.02$ $\boldsymbol{p = 0.083}$	$t = 2.50$ $p = 0.041$	$t = 2.53$ $p = 0.039$

with $\alpha = 0.05$, the testing of the **rgb2hsi** benchmark with a table of 256 elements and no update in case of collisions. For the other seven out of eight configurations, the *t-test* considers the data to be statistically different. Considering there are no other variables on the experiments other than how memoization is applied, we can confidently say the shown differences are caused by the technique.

5. Conclusions

This article presented a framework to automatically apply memoization to C/C++ applications. The resulting applications store outputs of pure functions mapped by their inputs. If these pure functions are called with the same inputs, the framework returns the stored value instead of recomputing it. Hence, this technique may lead to execution time and energy consumption improvements by simply avoiding unnecessary computations in scenarios where critical functions are called with repeating inputs. Our framework consists of two main components. First, a source-to-source compiler that modifies input C/C++ applications by targeting memoizable functions and replacing their calls with calls to wrappers that interface with a C memoization library. This library, the second component of our framework, maintains all the data structures and contains memoization logic needed to enhance applications with this optimization. The source-to-source compiler can be controlled by the user to perform analyses and decide which functions to target. We showed the impact of the memoization technique in four representative benchmarks and the usefulness of our software by measuring reductions in execution time and energy consumption.

Acknowledgments

This work was partially funded by the ANTAREX project through the EU H2020 FET-HPC program under grant no. 671623. Bispo acknowledges the

support provided by *Fundação para a Ciência e Tecnologia* (FCT), through the Postdoctoral scholarship SFRH/BPD/118211/2016.

References

[1] D. Michie, "memo" functions and machine learning, Nature 218 (5136) 19 (1968).

[2] C. Silvano, G. Agosta, J. Barbosa, A. Bartolini, A. R. Beccari, L. Benini, J. Bispo, J. M. P. Cardoso, C. Cavazzoni, S. Cherubin, R. Cmar, D. Gadioli, C. Manelfi, J. Martinovi, R. Nobre, G. Palermo, M. Palkovi, P. Pinto, E. Rohou, N. Sanna, K. Slaninov, The ANTAREX tool flow for monitoring and autotuning energy efficient HPC systems, in: 2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2017, pp. 308–316.

[3] A. Suresh, B. Narasimha Swamy, E. Rohou, A. Seznec, Intercepting Functions for Memoization: A Case Study Using Transcendental Functions, ACM Trans. Archit. Code Optim. 12 (2) (2015).

[4] A. Suresh, E. Rohou, A. Seznec, Compile-Time Function Memoization, in: 26th International Conference on Compiler Construction, Austin, TX, United States, 2017, pp. 45–54.

[5] J. M. Cardoso, J. G. Coutinho, T. Carvalho, P. C. Diniz, Z. Petrov, W. Luk, F. Gonçalves, Performance-driven instrumentation and mapping strategies using the lara aspect-oriented programming approach, Software: Practice and Experience 46 (2) 251–287 (2016).

[6] P. Michaud, Y. Sazeides, ATMI: analytical model of temperature in microprocessors, Third Annual Workshop on Modeling, Benchmarking and Simulation (MoBS) (2007).

Current code version

Nr.	Code metadata description	Please fill in this column
C1	Current code version	Clava: v3.0.13 Memoi: v1.0
C2	Permanent link to code/repository used for this code version	Clava: `https://github.com/specs-feup/clava` Memoi: `https://gforge.inria.fr/projects/memoization`
C3	Legal Code License	Clava: Apache License, 2.0 Memoi: LGPL V3
C4	Code versioning system used	Clava: git Memoi: svn
C5	Software code languages, tools, and services used	Clava: C++, Java, LARA Memoi: C, sed
C6	Compilation requirements, operating environments & dependencies	Linux, Windows, macOS
C7	If available Link to developer documentation/manual	
C8	Support email for questions	Clava: `joaobispo@gmail.com` Memoi: `loic.besnard@irisa.fr`

Table 4: Code metadata